**Imagen que contiene Logotipo

Descripción generada automáticamente**

**Hageo Juda Balam Méndez**

**2109012**

**1ª**

**Ingeniería en datos**

**Algorithms Fundamentals**

**Stages of program compiler and Levels of programming**

**Profesor Luis Gerardo Cámara**

**10/01/2022**

**Stages of program compiler**

The lexical analysis is the first phase of the compiler, starting by taking the pre-modified source code from some language processors which are written in the form of sentences, afterwards, the lexical analyzer breaks down these sentences into “tokens” (in programming language a token could be a keyword, an operator or a punctuation mark) by removing any whitespaces or comments in the source code; however, in order for a token to be valid it must be filtered with some predefined rules for each lexeme. These specifications are based on the grammar rules; summarizing the role of the lexical analyzer in the compiler design is to read the characters in the source code, check for valid tokens and pass the data to the syntax analyzer when it demands.

In the previous paragraph the syntax analyzer was mentioned as a different stage of the program compilation; nonetheless, each phase shares a built-in system called “Symbol table” which is an important structure included in the compiler which keeps track of the semantics variables and stores information about instances of various entities such as objects, classes, etc.

**Syntax analysis**

This is the phase where the grammar rules are verified in order to analyze the syntactical structure and check if the given input is in the appropriate syntax of the programming language, otherwise it will display error messages.

**Semantic analysis**

Us humans can easily understand the meaning and the context of words; however, this is not the case for machines, so the semantic analysis in simple terms draws a meaning from the text; for this third phase in of the compiler design which makes sure that the declarations or statements are semantically correct after the code have been processed by the previous steps in which the consistency of the code is tested, as any other part of the compiler this is crucial due to the next step is the code generation.

**Code generation**

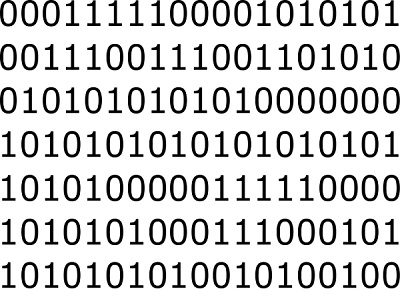
Once our code has been analyzed in the previous stages, we are almost good to go and run it since there is a couple compiles that generate the code twice, the first time generating the code in “intermediate language” like SIL, LLV, IR, MIR, CIL, etc. and then it proceeds to generate the code into a target language that is directly able to run or really close to in some cases; in this step we are aiming for a correct code that should be efficient.

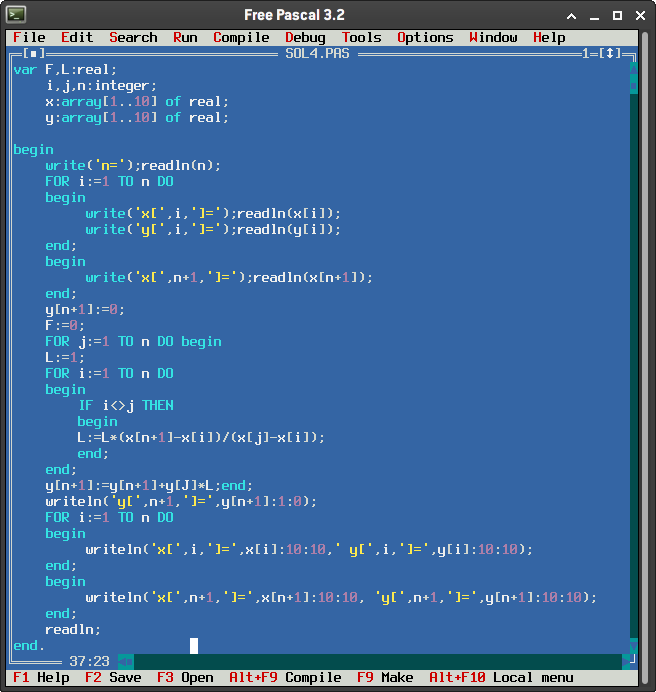
**Optimization**

Finally, we have optimization; the job of the optimizing compiler is to understand what your program does and essentially upgrade it so it can be smaller and faster.

**Levels of programming**

Every programming language has their own syntaxis and set of keywords that is used to give instructions and can be classified based on their level of abstraction, leaving us with Low-level language and High-level language.

The machine code takes part on the Low-level section, this language is also known as the elemental language of computers it is also composed of the binary numbers and it may loon like a simple but long sequence of zeros and ones; we are appreciating instructions that are read by the CPU, nowadays humans rarely deal directly with this language.

For the High-level languages we have FORTRAN, C, Pascal, etc. since they are far from machine languages and closer to human languages.

# **References**

Beal, V. (24 de 05 de 2021). *Token*. Recuperado el 10 de 01 de 2022, de Webopedia: https://www.webopedia.com/definitions/token/

Haoyi, L. (09 de 11 de 2019). *How an Optimizing Compiler Works*. Recuperado el 10 de 01 de 2022, de Haoyi's Programming Blog: https://www.lihaoyi.com/post/HowanOptimizingCompilerWorks.html

*Introduction to Code Generation*. (s.f.). Recuperado el 10 de 01 de 2022, de Loyola Marymount University Web Site: https://cs.lmu.edu/~ray/notes/codegen/

*Lexical Analysis*. (s.f.). Recuperado el 10 de 01 de 2022, de Tutorials Point: https://www.tutorialspoint.com/compiler\_design/compiler\_design\_lexical\_analysis.htm

Prajapati, S. (09 de 11 de 2021). *Symbol table compiler*. Recuperado el 10 de 01 de 2022, de GeeksforGeeks: https://www.geeksforgeeks.org/symbol-table-compiler/

*Program construction*. (s.f.). Recuperado el 10 de 01 de 2022, de BBC: https://www.bbc.co.uk/bitesize/guides/zmthsrd/revision/3

*Semantic Analysis in Compiler Design*. (22 de 04 de 2020). Recuperado el 10 de 01 de 2022, de GeeksforGeeks: https://www.geeksforgeeks.org/semantic-analysis-in-compiler-design/

Smith, J. (11 de 12 de 2021). *Syntax Analysis: Compiler Top Down & Bottom Up Parsing Types*. Recuperado el 10 de 01 de 2022, de Guru99: https://www.guru99.com/syntax-analysis-parsing-types.html#:~:text=Syntax%20Analysis%20is%20a%20second,the%20programming%20language%20or%20not.https://www.guru99.com/syntax-analysis-parsing-types.html#:~:text=Syntax%20Analysis%20is%20a%20second,the%20progr